WHESL: A Workflow Hierarchy-aware Fault Tolerance System

Subhendu Behera[1], Dong H. Ahn[2], Stephen Herbein[2], Frank Mueller[1], Barry L. Rountree[2]
[1]North Carolina State University ssbehera,fmuelle@ncsu.edu
[2]Lawrence Livermore National Laboratory anh1,herbein1,rountree4@llnl.gov

***Abstract*—Trends towards complex scientific workflows present unprecedented challenges to fault tolerance support in high-performance computing (HPC). While existing solutions such as checkpoint/restart (C/R) and resource over-provisioning work well at the application level, they do not scale to the demand for complex workflows. As workflows are composed of a large variety of components (e.g., simulations at different scales with data analysis and machine-learning programs), they must detect, propagate, and recover from a fault in a highly coordinated way, lest handling action itself can often do more harm than good. We propose Workflow Hierarchy-aware Exception Specification Language (WHESL), a novel solution that allows a modern workflow to specify and handle faults and exceptions among its disparate components in a coordinated and programmable manner. At the core of WHESL lies a domain-specific language that succinctly defines a rich set of temporal and spatial faults and exceptions and relates them to the hierarchy of workflow components. Our preliminary study using our prototype built on top of Flux, a next-generation hierarchical resource and job management system (RJMS), shows that WHESL can significantly extend the traditional HPC fault tolerance support for complex workflows.**

***Index Terms*—Fault Tolerance, Exception, HPC, Workflow, Dependency**

I. Introduction

Scientific workflows comprise a collection of dependent and often heterogeneous jobs, which may use different programming languages and execution models. They allow scientific programmers to express and automate complex simulations and computing, coupled multi-physics/multi-science applications and perform advanced data analyses [1]. A fault or exception in one of the components within these workflows can lead to a catastrophic failure of the whole workflow run, which then needs a re-run. Many fault-tolerance mechanisms (C/R and resource over-provisioning) do not take such complexity and dependencies into consideration. C/R applies to a single job within a workflow. However, any dependent secondary jobs within the workflow or other entities (such as scheduler, workflow manager) cannot coordinate to recover from an exception. Further, saving the state of the workflow or resource over-provisioning is not scalable at extreme-scale computing.

We propose a holistic approach to detection, propagation, and recovery based on the Multi-level Cooperative Exception Model [2]. Our developed framework performs node-level log analysis to detect temporal exceptions. Once detected, exceptions propagate through the workflow hierarchy based on the dependencies among relevant components of the workflow. During the propagation, exceptions can be transformed into another exception by an entity, which allows for spatial exception detection. For recovery, we promote a role-based exception handling mechanism that allows multiple entities to perform recovery from exceptions based on their role within the workflow in a coordinated manner. These specifications are processed by our framework in response to the user-provided WHESL exception description. Our framework is implemented as a service module within the Flux RJMS.

II. Design

A. Flux

Flux is a scalable, hierarchical RJMS that, through its ability to be nested within other schedulers and itself, produces high job submission and scheduling throughput for modern workflows. The consistent and a well-defined set of APIs allow users to address the challenges of complex co-scheduling and coordination of jobs and ensembles of small tasks within a workflow. Figure 1 depicts a Flux hierarchy instance. The communication framework implemented using ZeroMQ [3], a messaging library that provides common messaging patterns such as request-reply, publish-subscribe, etc., facilitates communication within the hierarchy.

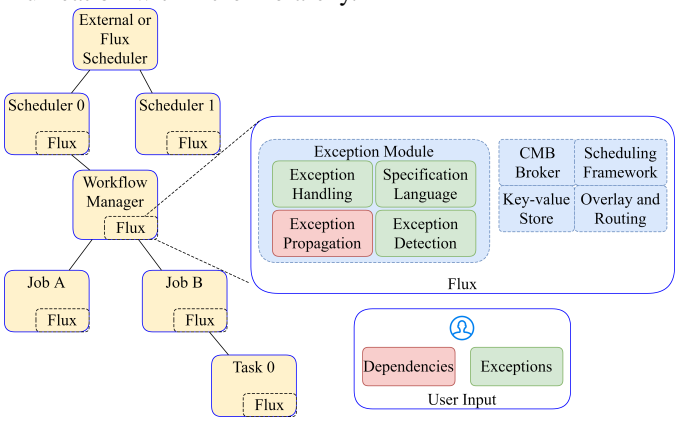

Fig. 1. Overall Architecture

B. WHESL

Figure 2 depicts the top-level rules of our Workflow Hierarchy-aware Exception Specification Language in Backus-Naur form. Our fault tolerance system is comprised of four major constructs, namely <component-list>, <workflow-roles>, <exception-list>, and <dependency-list>. Via WHESL, users can specify what components exist on a node, different types of workflow entities, exceptions along with their detection and handling procedures, as well as any dependencies between workflow components.

```
<system> -> { <component-list> } { <workflow-roles> } { <exception-list> } { <dependency-list> }

<component-list> -> <component> , <component-list> | EMPTY

<workflow-roles> -> <workflow-role> , <workflow-roles> | EMPTY
<workflow-role> -> TASK | JOB | WORKFLOW | DEPENDENT_TASK | DEPENDENT_JOB

<exception-list> -> <exception> ; <exception-list> | EMPTY
<exception> -> { ID; <origin-component>; <detection>; <handler-list> } //ID: exception name

<detection> -> { <location> , markers }
<location> -> STDOUT | STDERR
<markers> -> <marker-series> | <marker-timeseries>

<handler-list> -> { <handler> } ; <handler-list> | EMPTY
<handler> -> <workflow-role> , <source-exception> , <transformation-exception> , { <action-sequence> }
<action-sequence> -> <action-key> , <action-sequence> | EMPTY
<action-key> -> PUBLISH | WAIT | KILLTASK

<dependency-list> -> dependency, <dependency-list> | EMPTY
<dependency> -> { <source> ; <dependent>; <dependency-level> , <dependency-type>}
<source> -> { <jobid> , <startrank> , <endrank> }
<dependent> -> { <jobid> , <startrank> , <endrank> }
<dependency-type> -> ALL | TASK | <component> | <component-type>
<dependency-level> -> JOB | NODE
```

Fig. 2. Exception Specification Language

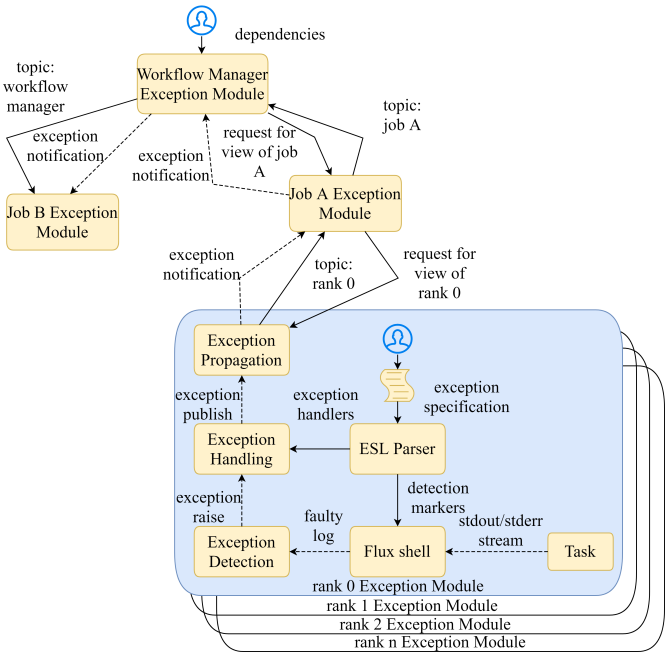

Fig. 3. Operational View

C. Detection, Propagation, and Recovery

Figure 3 depicts how our framework performs detection, propagation, and recovery of exceptions. The solid lines represent initialization operations while the dotted lines represent operations performed during run time. The Flux shell parses the output of stdout and stderr streams for the detection markers installed within it during initialization. Any faulty log that matches with a marker triggers a notification to the exception module. The exception module keeps track of the previously identified markers for all the exceptions and raises a temporal exception for the final detection marker.

The exception handling sub-module follows the exception specification provided by the user, and its role within the workflow determines the set of actions to handle the exception. The actions represented as a set of keys refer to predefined functions in Flux such as termination of a job or task, coordination through barriers. Exceptions are also republished to other subscribers as needed in the hierarchy for coordination, thereby creating a multi-cast propagation path. The transformation of an exception in the propagation path allows for spatial detection.

Our framework builds the exception propagation path as per the dependencies within the workflow. Figure 3 demonstrates the construction of a propagation path between jobs A and B. It also shows how exceptions flow from a node/rank within job A to B. The Workflow Manager adds the dependencies at the top of the hierarchy. Based on the dependency information, our propagation path construction algorithm performs a depth-first search to locate the source and creates a view (group) of components under it. A topic string represents the view, which is subscribed by the source's immediate parent. This view construction process is transitive (recursively applied to the workflow hierarchy). Hence, exceptions always follow the workflow hierarchy during propagation.

III. Case Study

We present a case study of node failure that can be detected by three different entities within a workflow. A job, which the failed node was part of, finds an unresponsive node during communication. The resource manager, Flux, detects it through a heartbeat timeout. Finally, if the whole job fails, then the workflow manager assumes a job failure. Without any coordination, each of them will start their recovery process, which may take contradictory or redundant actions leading to duplicate job or task submission. Also, any running dependent job will not be able to identify this situation and take measured action.

With WHESL, a peer node can detect the node failure exception as the Flux shell discovers the faulty log. The peer node's exception module will raise the exception and publish it to the job's exception module, which starts the recovery process. The recovery process may require a new node, which the Flux RJMS will allocate. If there are no spare nodes available, Flux can raise another exception to the Workflow Manager requesting an additional node. Further, the job's exception module publishes the node failure exception to a dependent job, which may need to throttle or pause its tasks until the recovery process is complete.

Acknowledgement

This research was supported by an appointment to the Lawrence Livermore National Laboratory's Compute Scholar Program. This work was also performed under the auspices of the U.S. Department of Energy by Lawrence Livermore National Laboratory under contract DE-AC52-07NA27344 (LLNL-POST-813928).

References

[1] E. Deelman, T. Peterka, I. Altintas, C. D. Carothers, K. K. van Dam, K. Moreland, M. Parashar, L. Ramakrishnan, M. Taufer, and J. Vetter, "The future of scientific workflows," *Int. J. High Perform. Comput. Appl.*, vol. 32, no. 1, p. 159–175, Jan. 2018.

[2] S. Herbein, D. Domyancic, P. Minner, I. Laguna, R. Ferreira da Silva, and D. Ahn, "Mcem: Multi-level cooperative exception model for hpc workflows," 06 2019, pp. 27–32.

[3] P. Hintjens. Zeromq: An open-source universal messaging library. [Online]. Available: http://zguide.zeromq.org/page:all